Notes on type-driven development with Idris

Christiano Braga

DRAFT

Abstract

In these notes we explore the type-driven software development approach. Essentially, it relies on the concept of dependent types to enforce safety behavior. Idris is our programming language of choice.

DRAFT

Contents

DRAFT

Chapter 1

The need for types

This section motivates the use of strong typing with a very very simple example: Bhaskara's theorem. In a tutorial way, we illustrate how types are necessary and, more specifically, how Idris' strong-typing presents itslef as a powerful development tool.

From school: Bhaskara's theorem[1]

$$ax^2 + bx + c = 0 \quad \Rightarrow \quad x = \frac{-b \pm \sqrt{\delta}}{2a}$$
$$\text{where } \delta = b^2 - 4acb$$

As functions:

$$\begin{aligned}
\texttt{bhask}(a,b,c) &= \left(-b + \sqrt{\texttt{delta}(a,b,c)}/2a, -b - \sqrt{\texttt{delta}(a,b,c)}/2a\right) \\
\texttt{delta}(a,b,c) &= b^2 - 4acb
\end{aligned}$$

1.1 First attempt: no types.

In Python:

```
from math import sqrt

def delta(a,b,c):
    return  (b * b) - (4 * a * c)

def bhask(a,b,c):
    d = delta(a,b,c)
    sr = sqrt(d)
    r1 = (-b + sr) / 2 * a
    r2 = (-b - sr) / 2 * a
    return (r1, r2)
```

When we run `bhask(1,2,3)` the following is spit out:

[1] For solving 2^{nd} degree polynomials. But this could might as well be an Excel formula, for instance! I mention Excel because that Microsoft is devoting serious efforts to develop a type system for Excel.

DRAFT

```
Traceback (most recent call last):
  File "bhask.py", line 16, in <module>
    bhask(1,2,3)
  File "bhask.py", line 9, in bhask
    sr = sqrt(d)
ValueError: math domain error
```

This cryptic answer is only because we rushed into a direct implementation and forgot that `delta(a,b,c)` may return a *negative* value!

1.2 Second attempt: still no types.

Now, assuming we are instered only on Real results, how should `bhask` deal with the possilibity of a negative `delta`?

One possibilty is to raise an *exception*:

```
from math import sqrt

def delta(a,b,c):
    return  (b * b) - (4 * a * c)

def bhask(a,b,c):
    d = delta(a,b,c)
    if d >= 0:
        sr = sqrt(d)
        r1 = (-b + sr) / 2 * a
        r2 = (-b - sr) / 2 * a
        return (r1, r2)
    else:
        raise Exception("No Real results.")
```

This implementation gives us a more *precise* answer:

```
Tue Jul 30@17:18:02:sc$ python3 -i bhask.py
Traceback (most recent call last):
  File "bhask.py", line 16, in <module>
    bhask(1,2,3)
  File "bhask.py", line 14, in bhask
    raise Exception("No Real results.")
Exception: No Real results.
```

A very **important** point here is that we only find all this out while actually *running* our implementation. Can't we do better? That is, let the **compiler** find out that `delta` may become a negative number and complain if this is not properly handled?

1.3 In Idris.

Let us play with `delta` first.

Strongly-typed languages, such as Idris, force us to think about types right away as we need to define `delta`'s signature. If we make the same mistake we did in the first attempt and forget that `delta` may become negative, we may write,

DRAFT

```
> delta : (a :  Nat) -> (b : Nat) -> (c : Nat) -> Nat
> delta a b c = (b * b) - (4 * a * c)
```

the compiler would tell us:

```
Type checking ./intro.lidr
intro.lidr:100:26:
    |
100 | > delta a b c =  (b * b) - (4 * a * c)
    |                          ^
When checking right hand side of delta with expected type
        Nat

When checking argument smaller to function Prelude.Nat.-:
        Can't find a value of type
                LTE (mult (plus a (plus a (plus a (plus a 0)))) c) (mult b b)
```

This is cryptic, in a first-glance, but tells us precisely **what** is wrong **and** at **compile** time. The problem is **with subtraction**: the type checker was not able to solve the inequality, defined in Idris' libraries,

$$4ac \leq b^2$$

in order to produce a **natural** number while computing `delta`, as natural numbers can not be negative!

1.3.1 First fix.

And we have not even started thinking about `bhask` yet! But let us first make `delta` type right by changing its signature:

```
delta : (a :  Nat) -> (b : Nat) -> (c : Nat) -> Int
delta a b c =  (b * b) - (4 * a * c)
```

To see the effect of this change, load `delta-fix.lidr` with the command:

```
:l delta-fix.lidr
```

Don't be so happy though! This is not what we want yet.

```
Type checking ./delta-fix.lidr
delta-fix.lidr:5:18-38:
  |
5 | > delta a b c =  (b * b) - (4 * a * c)
  |                  ~~~~~~~~~~~~~~~~~~~~~
When checking right hand side of delta with expected type
        Int

Can't disambiguate since no name has a suitable type:
        Prelude.Interfaces.-, Prelude.Nat.-

Holes: Main.delta
```

Idris does not know which subtraction operation to use because we are operating operating with natural numbers but we should return an integer! A casting is in order!

1.3.2 Second fix.

Think about why we should cast the right-hand side expression in the following way:

```
delta : (a :  Nat) -> (b : Nat) -> (c : Nat) -> Int
delta a b c =  (cast (b * b)) - (cast (4 * a * c))
```

and not the whole right-hand side of `delta` at once.

To see the effect of this change, load `delta-fix2.lidr` with the command:

```
:l delta-fix2.lidr
```

You should finally be able to see

```
Type checking ./delta-fix2.lidr
*delta-fix2>
```

and run `delta 1 2 3`, for instance, to see the following result.

```
*delta-fix2> delta 1 2 3
-8 : Int
```

Your session should look like this at this point:

```
Wed Jul 31@14:22:57:sc$ idris intro.lidr
     ____    __     _
    /  _/___/ /____(_)____
    / // __  / ___/ / ___/     Version 1.3.2
  _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
 /___/\__,_/_/  /_/____/       Type :? for help

Idris is free software with ABSOLUTELY NO WARRANTY.
For details type :warranty.
Type checking ./intro.lidr
intro.lidr:125:25:
    |
125 | > delta a b c = (b * b) - (4 * a * c)
    |                         ^
When checking right hand side of delta with expected type
        Nat

When checking argument smaller to function Prelude.Nat.-:
        Can't find a value of type
                LTE (mult (plus a (plus a (plus a (plus a 0)))) c) (mult b b)

Holes: Main.delta
*intro> :l delta-fix.lidr
Type checking ./delta-fix.lidr
delta-fix.lidr:5:18-38:
  |
5 | > delta a b c =  (b * b) - (4 * a * c)
  |                  ~~~~~~~~~~~~~~~~~~~~~
When checking right hand side of delta with expected type
        Int

Can't disambiguate since no name has a suitable type:
        Prelude.Interfaces.-, Prelude.Nat.-
```

DRAFT

```
Holes: Main.delta
*delta-fix> :l delta-fix2.lidr
*delta-fix2> delta 1 2 3
-8 : Int
*delta-fix2>
```

1.3.3 Bhaskara at last!

Painful, no?

No!

The compiler is our *friend* and true friends do not always bring us good news! Think about it using this metaphor: do you prefer a shallow friend, such as Python, that says yes to (almost) everything we say (at compile time), but is not there for us when we really need it (at run time), or a *true* friend, such as Idris, that tells us that things are not all right all the time, but is there for us when we need it?

Another way to put it is that "With great power comes great rsponsibility!", as the philosopher Ben Parker used to say... Strong typing, and in particular this form of strong typing, that relies on *automated theorem proving* requires some effort from our part in order to precisely tell the compiler how things should be.

Having said that, let us finish this example by writing `bhask` function.

1.3.3.1 Bhaskara: first attempt

Bhaskara's solution for second-degree polynomials gives no Real solution (when $\delta < 0$), one (when $\delta = 0$), or two (when $\delta > 0$). Since "The Winter is Coming" we should be prepared for two roots:

```
bhask : (a :  Nat) -> (b : Nat) -> (c : Nat) -> (Double, Double)
bhask a b c = ((-b + (sqrt (delta a b c))) / (2 * a),
               (-b - (sqrt (delta a b c))) / (2 * a))
```

Moreover, we should now work with the Idris `Double` type, because of the `sqrt` function. Run

```
*bhask-fun> :t sqrt
sqrt : Double -> Double
```

Again, our naivete plays a trick on us:

```
Type checking ./bhask-fun.lidr
bhask-fun.lidr:2:19:
  |
2 | > bhask a b c = ((-b + (sqrt (delta a b c))) / (2 * a),
                     (-b - (sqrt (delta a b c))) / (2 * a))
  |                   ^
When checking right hand side of bhask with expected type
        (Double, Double)

When checking an application of function Prelude.Interfaces.negate:
        Type mismatch between
                Nat (Type of b)
        and
                Double (Expected type)
```

DRAFT

Load file `bhask-fun.lidr` to see this effect.

We should write `negate b` instead of `- b`, as `-` is a *binary* operation only in Idris. Moreover, we should *not* be able to negate a natural number! Again, casting is necessary.

Let us fix all casting problem at once, the final definitions should be as follows:

```idris
delta : (a :  Nat) -> (b : Nat) -> (c : Nat) -> Int
delta a b c =  (cast (b * b)) - (cast (4 * a * c))
bhask : (a :  Nat) -> (b : Nat) -> (c : Nat) -> (Double, Double)
bhask a b c =
  (negate (cast b) + (sqrt (cast (delta a b c))) / cast (2 * a),
   negate (cast b) - (sqrt (cast (delta a b c))) / cast (2 * a))
```

We can now play with `bhask`, after executing `:l bhask-fun-fix.lidr`

```idris
Type checking ./bhask-fun-fix.lidr
*bhask-fun-fix> bhask 1 10 4
(-5.41742430504416, -14.582575694955839) : (Double, Double)
*bhask-fun-fix> bhask 1 2 3
(NaN, NaN) : (Double, Double)
```

Note that when $\delta < 0$ Idris gives a `NaN` value, which stands for *Not a number*. In other words, `bhask` is **total** as opposed to the **partial** approach in Python where we needed to raise an exception to capture the situation where the roots are not Real numbers.

Idris can help us identify when a function is total. We simply need to run:

```idris
*bhask-fun-fix> :total bhask
Main.bhask is Total
```

1.4 Wrapping-up

Our intentions in this chapter are manifold:

1. First and foremost motivate strong-typing in Idris.
2. Introduce notation for functions in Idris:
3. The signature of a function, sych as `delta`: - A function declaration has a name, formal parameters and a return type, such as `delta`.
   `delta : (a : Nat) -> (b : Nat) -> (c : Nat) -> Int` - Currying The formal parameters of a function are declared using the so-called Currying form (after Haskell Curry): currying is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.
   This allows to *partially apply* a function! For instance, we can call `delta 1 2`. This will produce a function that expects a number and then behaves as `delta`.

~~~
Take a look at the following session:
```idris
*bhask-fun-fix> delta
delta : Nat -> Nat -> Nat -> Int
*bhask-fun-fix> delta 1
delta 1 : Nat -> Nat -> Int
*bhask-fun-fix> delta 1 2
delta 1 2 : Nat -> Int
*bhask-fun-fix> delta 1 2 3
-8 : Int
*bhask-fun-fix> (delta 1) 2
~~~

DRAFT

~~~~
delta 1 2 : Nat -> Int
*bhask-fun-fix> ((delta 1) 2) 3
-8 : Int
```
At the end of the day, `delta 1 2 3` is just _syntax sugar_ for `((delta 1) 2) 3`.
~~~~

1. Total functions. From Idris' FAQ:

   I have an obviously terminating program, but Idris says it possibly isn't total. Why is that?

   Idris can't decide in general whether a program is terminating due to the undecidability of the Halting Problem. It is possible, however, to identify some programs which are definitely terminating. Idris does this using "size change termination" which looks for recursive paths from a function back to itself. On such a path, there must be at least one argument which converges to a base case.

   - Mutually recursive functions are supported
   - However, all functions on the path must be fully applied. In particular, higher order applications are not supported
   - Idris identifies arguments which converge to a base case by looking for recursive calls to syntactically smaller arguments of inputs. e.g. k is syntactically smaller than $S(Sk)$ because k is a subterm of $S(Sk)$, but (k, k) is not syntactically smaller than (Sk, Sk).

2. Type casting. We have used `cast` many times in order to *inject* our values from one type into another.

3. Some Read-Eval-Print-Loop (REPL) commands. We have seen how to load a file with `:l`, check its type with `:t`, and check weather a function is total or not with `:total`.

DRAFT

Chapter 2

The need for dependent types

Overflow conditions in software appear to be a simple thing to implement. An important counter-example is the Ariane 5 rocket that exploded due to a down cast from 64-bit number into a 16-bit one.

> The Ariane 5 had cost nearly $8 billion to develop, and was carrying a $500 million satellite payload when it exploded.

11 of the most costly software errors in history

In this chapter we look at a simpliflied version of the `Vector` datatype, avaiable in Idris' library, to try and understand how *dependent typing* can be useful to have type-safe array handling that could help prevent catastrophes such as the Ariane 5 explosion.

2.1 `Vector`

A datatype is nothing but an implementation of some "domain of information". It could very well represent low level information such as data acquired by a sensor in a Internet of Things (IoT) system or the structure that organizes the decision making process in planning.

Our datatype here is quite simple but illustrates very well how dependent types may help safe data modeling and implementation.

```
> module Vect
> data Vect : Nat -> Type -> Type where
>     Nil  : Vect Z a
>     (::) : (x : a) -> (xs : Vect k a) -> Vect (S k) a
```

An array or vector is built or *constructed* using either one of the constructor operations (unary) `Nil` or (binary) `::`. (The `module` keyword here simply defines a *namespace* where `Vect` will live.) After loading this file in Idris you could try

```
*tnfdt> 1 :: Vect.Nil
[1] : Vect 1 Integer
```

at the REPL.

This says that the term `[1]` has type `Vect 1 Integer` meaning that it is a vector with one element and that its elements of the `Integer` type, Idris' basic types.

Maybe this is a lot to take! *Just breath* and let us think about it for a moment.

DRAFT

Types are defined in terms of constructor operators. This means that an *instance* of this type is written down as `1 :: Vect.Nil`. In a procedural language you could write it with a code similar to

```
v = insert(1, createVect(1))
```

where `createVect` returns a vector of a given size and `insert` puts an element on the given vector. The point is that we usually create objects or allocate memory to represent data in variables (so called *side effects*) while in functional programming we *symbolically* manipulate them, as in the example above.

This is a major paradigm-shift for those not familiar with functional programming. Be certain that it will become easier as time goes by, but let's move on!

2.2 Dependency

Let's look at the instance first and then to the type declaration. Note that the type of `[1]` is `Vect 1 Integer`. The type of a Vect *depends* on its *size*! Think about examples of vectors in programming languages you know. If you query for the type of a given vector, if at all possible, what the run-time of your programming language will answer?

In Python, for instance, you would get something like,

```
v = [1,2,3]
type(v)
<class 'list'>
```

that is, is a `list` and that's all! In C an array is a pointer! (A reference to a memory address, for crying out loud!)

In Idris, we know it is a vector and its size, an important property of this datatype. Cool! And so what?

We can take advantage of that while programming. We could write a function that does *not*, under no circumstances, goes beyond the limits of a vector, that is, index it beyond its range!

2.2.0.1 The `zip` function

The `zip` function simple creates pairs of elements out of two instances of `Vect` *with the same size*. Here is what it look like:

```
> zip : Vect n a -> Vect n b -> Vect n (a, b)
> zip Nil Nil = Nil
> zip (x :: xs) (y :: ys) = (x, y) :: zip xs ys
```

What on earth is it? Do you remember how to declare a function in Idris? Well, is pretty-much that. The difference here is that we are now programming with *pattern matching*. And what is it? Simply define a function by *cases*. When we hit an instance of `Vect`, how does it look like? It is either the empty vector, built with constructor `Nil`, or a non-empty vector, built using operator `::`. These two cases are represented by each equation above. The first equation declares the case of "zipping" two *empty* vectors and the second one handles two *non-empty* vectors, specified by the *pattern* `x :: xs`, that is, a vector whose first element is `x` and its remaining elements are represented by a (sub)vector `xs`.

For instance, if we could write

```
*tnfdt> Vect.zip [1,2,3] ["a", "b", "c"]
[(1, "a"), (2, "b"), (3, "c")] : Vect 3 (Integer, String)
```

DRAFT

and get the expected vector of pairs produced by `zip`. (I used `Vect.zip` only because there are other `zip` functions coming from Idris' standard library.) Note that the type of `[(1, "a"), (2, "b"), (3, "c")]` is `Vect 3 (Integer, String)` where 3 is the size of the vector and `(Integer, String)`, denoting pairs of integers and strings, is the type of the elements of vector that `zip` calculates.

Note some additional interesting things about `zip`'s declaration: - The signature of `zip` is `zip : Vect n a -> Vect n b -> Vect n (a, b)`. The variable `n` here stands for the size of the vector. Variables `a` and `b` denote the types of the elements of the vectors being zipped. That is, the `Vect` type is *generic*, as the type of its elements are underspecified, and is *dependent* on the **number** denoting its size. Again, `n` is a *number*, and `a` (or `b`, for that matter) is a *type*!

Now, take a look at this:

```
*tnfdt> Vect.zip [1,2,3] ["a", "b"]
(input):1:19-21:When checking argument xs to constructor Vect.:::
        Type mismatch between
                Vect 0 a (Type of [])
        and
                Vect 1 String (Expected type)

        Specifically:
                Type mismatch between
                        0
                and
                        1
```

What does this mean? This is a *type checking* error, complaining about an attempt to zip vectors of different sizes. This is *not* an exception, raised while trying to execute `zip`. This is a *compile* type message, regarding the case of zip a vector of length 1 (the last element of the first vector), and a 0-sized vector (from the second vector).

In Idris, types can be manipulated just like any other language construct.

2.2.1 Conclusion.

Ariane 5 would not have exploded (from the bit conversion perspective) if the function that accidentally cast a 64-bit vector into a 16-bit one was written with this approach.

2.3 Wrapping-up

1. Defining datatypes.
2. Defining dependent datatypes.
3. Using dependent datatypes to find errors at compile time.
4. Type expressions.

DRAFT

Chapter 3

Programming with type-level functions

DRAFT

Here are a couple of examples where first-class types can be useful: - Given an HTML form on a web page, you can calculate the type of a function to process inputs in the form. - Given a database schema, you can calculate types for queries on that database. In other words, the type of a value returned by a database query may vary *depending on* the database *schema* and the *query* itself, calculated by **type-level functions**.

This should be useful in a number of contexts such as Data validation in Robotic Process Automation, SQL Injection, (Business) Process Protocol Validation, just to name a few.

In this section we discuss and illustrate how this way of programming is available in the Idris language.

3.1 Formatted output example

This examples explores some of the components for the RPA scenario. It explores how to make strings from properly-typed data using type-functions.

```
> module Format
>
> data Format =
> Number Format
> | Str Format
> | Lit String Format
> | End
```

Try this at the REPL:

```
*pwfct> Str (Lit " = " (Number End))
Str (Lit " = " (Number End)) : Format
```

```
> PrintfType : Format -> Type
> PrintfType (Number fmt) = (i : Int) -> PrintfType fmt
> PrintfType (Str fmt) = (str : String) -> PrintfType fmt
> PrintfType (Lit str fmt) = PrintfType fmt
> PrintfType End = String
```

Try this at the REPL:

```
*pwfct> PrintfType (Str (Lit " = " (Number End)))
String -> Int -> String : Type
```

```
> printfFmt : (fmt : Format) -> (acc : String) -> PrintfType fmt
> printfFmt (Number fmt) acc = \i => printfFmt fmt (acc ++ show i)
> printfFmt (Str fmt) acc = \str => printfFmt fmt (acc ++ str)
> printfFmt (Lit lit fmt) acc = printfFmt fmt (acc ++ lit)
> printfFmt End acc = acc
```

```
> toFormat : (xs : List Char) -> Format
> toFormat [] = End
> toFormat ('%' :: 'd' :: chars) = Number (toFormat chars)
> toFormat ('%' :: 's' :: chars) = Str (toFormat chars)
> toFormat ('%' :: chars) = Lit "%" (toFormat chars)
> toFormat (c :: chars) = case toFormat chars of
>                              Lit lit chars' => Lit (strCons c lit) chars'
>                              fmt => Lit (strCons c "") fmt
```

```
> printf : (fmt : String) -> PrintfType (toFormat (unpack fmt))
> printf fmt = printfFmt _ ""
```

Try this out at the REPL:

```
*pwfct> :let msg = "The author of %s, published in %d, is %s."
*pwfct> :let b = "A Brief History of Time"
*pwfct> :let a = "Stephen Hawking"
*pwfct> :let y = the Int 1988
*pwfct> printf msg b y a
"The author of A Brief History of Time, published in 1988, is Stephen Hawking." : String
```

For variable `y` we had to make sure it is an `Int` (finite), not an `Integer` (infinite) number, due to `PrintfType` definition. This is what `the Int 1988` does. Try it without the casting and see what happens…

3.2 Caveats

(From TDD book.)

In general, it's best to consider type-level functions in exactly the same way as ordinary functions. This isn't always the case, though. There are a couple of technical differences that are useful to know about:

- Type-level functions exist at *compile* time only. There's no runtime representation of Type, and no way to inspect a Type directly, such as pattern matching.
- Only functions that are total will be evaluated at the type level. A function that isn't total may not terminate, or may not cover all possible inputs. Therefore, to ensure that type-checking itself terminates, functions that are not total are treated as constants at the type level, and don't evaluate further.

DRAFT